Register Allocation by Optimal Graph Coloring

Christian Andersson

Dept. of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
chrisand@cs.lth.se

Abstract. We here present new insights of properties of real-life interference graphs emerging in register allocation. These new insights imply good hopes for a possibility of improving the coloring approach towards optimal solutions. The conclusions are based on measurements of nearly 28,000 real instances of such interference graphs. All the instances explored are determined to possess the so-called 1-perfectness property, a fact that seems to make them easy to color optimally. The exact algorithms presented not only produce better solutions than the traditional heuristic methods, but also, indeed, seem to perform surprisingly fast, according to the measurements on our implementations.

1 Introduction

For almost all architectures register allocation is among the most important of compiler optimizations. Computations involving only register operands are much faster than those involving memory operands. An effective utilization of the limited register file of the target machine may tremendously speed up program execution, compared to the same program compiled with a poor allocation.

Graph coloring is an elegant approach to the register allocation problem. Traditional algorithms used by compilers today [4,5,3,9,17] make use of approximate heuristics to accomplish the colorings.

Here we do not propose a new algorithm for register allocation. Our experiments, however, suggest that such an algorithm may well be designed, which guarantees optimal colorings for the purpose of a good allocation. Despite the fact that graph coloring is an NP-complete problem, the input graphs in the case of register allocation certainly seem to be efficiently colored, even when using an exact algorithm.

2 Background

Let $V = \{v_1, v_2, v_3, \ldots\}$ be the set of variables in a given intermediate representation (IR) of a program. Given a certain point p in the program flow, a variable $v_i \in V$ is said to be *live* if it is defined above p but not yet used for the last time. A *live range* (LR) for a variable $v_i \in V$ is a sequence of instructions beginning with the definition of v_i and ending with the last use of v_i. An *LR interference*

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 33–45, 2003.
© Springer-Verlag Berlin Heidelberg 2003

is a pair $\langle\cdot,\cdot\rangle$ of variables whose live ranges intersect. Variables involved in such an interference can not be assigned to the same register. We denote by E the set of LR interference pairs. The *register allocation problem* is the problem of finding a mapping $c : V \mapsto \{r_1, r_2, \ldots, r_k\}$, where r_i are the registers of the target machine, such that $k \leq N$, where N is the total number of registers available and such that $\langle v_i, v_j\rangle \in E \Rightarrow c(v_i) \neq c(v_j)$. This corresponds closely to the well-known Graph-Coloring problem of finding a k-coloring of the graph $G = (V, E)$, which we call the *interference graph* (IG).

2.1 Graph Coloring

More exactly the Graph-Coloring problem is to determine for a given graph G and a positive integer k whether there exists a proper k-coloring. The smallest positive integer k for which a k-coloring exists is called the *chromatic number of* G, which is denoted by $\chi(G)$. Graph-Coloring is NP-complete [8].

The coloring problem seems not only practically impossible to solve exactly in the general case. Numerous works in this field from the past decades show that it is very hard to find algorithms that give good approximate solutions without restricting the types of input graphs. One well-known and obvious lower bound on the chromatic number $\chi(G)$ is the *clique number*, which is denoted by $\omega(G)$. A *clique* Q in a graph $G = (V, E)$ is a subset of V such that the subgraph G' induced by Q is *complete*, i.e., a graph in which all vertices are pairwise adjacent, and hence have to be colored using no less than $|Q|$ colors. The Maximum-Clique problem asks for the size of the largest clique of a given graph, the solution of which is the *clique number* $\omega(G)$. There are, however, two problems with this lower bound:

1. Maximum-Clique is (also) NP-complete [6].
2. According to, e.g., Kučera the gap between the clique number ω and the chromatic number χ is usually so large, that ω seems not to be usable as a lower bound on χ [14].

2.2 Traditional Approaches

Since Graph-Coloring is NP-complete [8], traditional register allocation implementations [4,5,3,9,17] rely on an approximate greedy algorithm for accomplishing the colorings. The technique used in all these implementations is based on a simple coloring heuristic [12]:

If $G = (V, E)$ contains a vertex v with a degree $\delta(v) < k$, i.e., with fewer than k neighbors, then let G' be the graph $G - \{v\}$, obtained by removing v, i.e., the subgraph of G induced by $V \setminus \{v\}$. If G' can be colored, then so can G, for when reinserting v into the colored graph G', the neighbors of v have at most $k - 1$ colors among them. Hence a free color can always be found for v.

The reduction above is called the *simplify* pass. The vertices reduced from the graph are temporarily pushed onto a stack. If, at some point during simplification, the graph G has vertices only of *significant degree*, i.e., vertices v of

degree $\delta(v) \geq k$, then the heuristic fails and one vertex is marked for *spilling*. That is, we choose one vertex in the graph and decide to represent it in memory, not registers, during program execution. If, during a simplify pass, one or more vertices are marked for spilling, the program must be rewritten with explicit loads and stores, and new live ranges must be computed. Then the simplify pass is repeated. This process iterates until *simplify* succeeds with no spills. Finally, the graph is rebuilt, popping vertices from the stack and assigning colors to them. This pass is called *select*.

3 Interference Graph Characterization

The traditional methods described above, which are used in register allocation algorithms today, are approximate. Our experiments show, however, that making optimal colorings using exponential algorithms, may actually be a possible way of coloring graphs in the register allocation case. The key to this conjecture is the claimed so-called 1-perfectness of interference graphs.

3.1 Graph Perfectness

In the study of the so-called Shannon capacity of graphs, László Lovász in the 1970's introduced the ϑ-function, which has enjoyed a great interest in the last decades. For instance, its properties constitute the basis of a later on proven fact, that there are important instances of graphs (the so-called *perfect* graphs), whose possible k-colorability can indeed be determined in polynomial time.

The $\vartheta(G)$ function has two important and quite remarkable properties [11]:

1. $\omega(G) \leq \vartheta(\overline{G}) \leq \chi(G)$ [1] (The Sandwich Theorem)
2. For all graphs G, $\vartheta(G)$ is computable in polynomial time.

Those special instances of graphs G for which $\omega(G') = \chi(G')$ holds for each induced subgraph G' are said to be *perfect*, and they are indeed perfect in that particular sense that their possible k-colorability can be determined in polynomial time, as a direct consequence of the above properties. There are, however, no (or at least very few) proposals of algorithms which use this fact, and which run efficiently in practice. Moreover, the status of the recognition problem of the class of all perfect graphs is unknown.

Despite the fact that nobody has succeeded in designing an algorithm that efficiently solves the polynomial problem of perfect graphs, the elegance of the theory of these special instances makes it interesting to explore the possible perfectness of the interference graphs occurring in register allocation.

[1] The *complement graph of* $G = (V, E)$ is the graph $\overline{G} = (V, \overline{E})$, where

$$\overline{E} = \left\{ e = \langle u, v \rangle \,\middle|\, u, v \in V,\ u \neq v, \text{and}\ \langle u, v \rangle \notin E \right\}.$$

Furthermore, Olivier Coudert, who works in the field of logic synthesis and verification, wrote in 1997 a very interesting paper [7], on the claimed simplicity of coloring "real-life" graphs, i.e., graphs which occur in problem domains such as VLSI CAD, scheduling, and resource binding and sharing. This simplicity is, according to Coudert, basically a consequence of the fact that most of the graphs investigated are *1-perfect*, i.e., they are graphs such that $\omega(G) = \chi(G)$, however, not necessarily for all subgraphs as in the case of perfect graphs.

4 Interference Graph Experiments

The input graphs we have used for our metrics come from two sources:

- Andrew W. Appel and Lal George have published a large set of interference graphs [1] generated by their compiler for Standard ML of New Jersey, compiling itself. The 27,921 actual LR interference graphs differ in size from around 25 vertices and 200 edges up to graphs with 7,500 vertices and 70,000 edges. These graphs do not, however, constitute the data used in empirical measurements reported by the authors in their articles on Iterated Register Coalescing [9,10].
- In a project task in the Lund University course on optimizing compilers[2], an SSA based experimental lab compiler for a subset of the C programming language is provided, in which students are to implement optimization algorithms. The best contribution in the fall 2000 course was provided by Per Cederberg, PhD candidate at the Division of Robotics, Department of Mechanical Engineering, Lund University[3]. His implementation included, for instance, algorithms for constant and copy propagation, dead code elimination, global value numbering, loop-invariant code motion and the register allocation algorithm proposed by George and Appel. Cederberg kindly let us use his implementation for our experiments.

When looking at the interference graphs we have had access to, they indeed seem to demonstrate some characteristics that point towards their potential 1-perfectness. For example, interference graphs tend not to be very dense, although they have large cliques. In other words, the density of these graphs seems not to be uniformly spread out over the whole graph, but rather localized to one or a few "clique-like" parts. Such characteristics certainly suggest a possibility of 1-perfectness, and make it interesting to investigate whether Coudert's conjecture can or cannot be confirmed for interference graphs.

In order to decide whether a graph G is 1-perfect, we need to solve two NP-hard problems (as far as we know today), the Graph-Coloring problem and the Maximum-Clique problem. Our only possibility is to implement exact algorithms, i.e., an approximate solution to either of these problems is not adequate.

[2] `http://www.cs.lth.se/Education/Courses/EDA230/`
[3] `http://www.robotics.lu.se/`

The algorithms for exactly solving both of these problems are fairly simple and well-known. The simplicity of the algorithm does, however, not imply that they are fast — both of them obviously have exponential worst case execution time, since we may need to perform an exhaustive search to find the optimal solution.

4.1 Sequential Coloring

Algorithm 1 shows a backtracking search for an optimal coloring of an input graph G. By initially searching for the maximum clique Q we get not only a lower bound ω on the chromatic number χ, but also a partial coloring of the graph. (The vertices of Q are colored using ω colors, which is optimal since all those vertices are pairwise adjacent in G.) If we are lucky, the graph is 1-perfect, and when the recursive part of the algorithm finds a coloring using ω colors, the search can be interrupted.

Algorithm 1 Create an optimal proper coloring of a graph G using a standard backtracking search algorithm. Return the chromatic number $\chi(G)$, and leave the coloring in a map *color*, indexed by vertices.

```
SEQUENTIAL-COLOR(G)
 1  Q ← MAXIMUM-CLIQUE(G)
 2  k ← 0
 3  foreach v ∈ Q do
 4      k ← k + 1
 5      color[v] ← k
 6  return SEQUENTIAL-COLOR-RECURSIVE(G, k, |V(G)| + 1, |Q|)
```

Input: G is a graph, partially colored using k colors. χ is the current value on the chromatic number and ω is the lower bound given by MAXIMUM-CLIQUE.

```
SEQUENTIAL-COLOR-RECURSIVE(G, k, χ, ω)
 1  if G is entirely colored then
 2      return k
 3  v ← an uncolored vertex of G
 4  foreach c ∈ [1,MIN(k + 1, χ − 1)] do
 5      if ∀n ∈ N[v], color[n] ≠ c then        ▷ N[v] is the neighborhood of v
 6          color[v] ← c
 7          χ ← SEQUENTIAL-COLOR-RECURSIVE(G, MAX(c, k), χ, ω)
 8          if χ = ω then                       ▷ 1-perfect graph
 9              return χ
10  return χ                                    ▷ result after an exhaustive search
```

If, on the other hand, the graph is not 1-perfect, the maximum clique calculation will not be of any help at all. The algorithm then has to exhaustively enumerate all potential colorings that would improve on the chromatic number, which can take exponential time. The problem is that the lower bound is static in the sense

that it is not reevaluated at each recursion. Moreover, the algorithm simultaneously uses several *unsaturated* colors. (A color c is said to be *saturated* if it can not be used anymore to extend a partial coloring.) Efficiently estimating a lower bound on the number of colors necessary to complete an unsaturated coloring is an open problem.

In [7] it is concluded that the maximum clique is tremendously important when coloring 1-perfect graphs. If the clique found is not maximal, we are left with the same problem as when trying to color graphs which are not 1-perfect.

One important part of the algorithm is left unspecified: In what order should the uncolored vertices be picked? Kučera showed in 1991, that it is practically impossible to find an ordering which performs well in the general case when using a greedy approach to the coloring problem [15].

The ordering of the vertices in the exact sequential algorithm may, however, have a strong impact on the efficiency of the search. Brélaz in 1979 proposed an efficient method called the DSATUR heuristic [2]. It consists of picking the vertex that has the largest saturation number, i.e., the number of colors used by its neighbors. If implementing the algorithm carefully, the information on vertex saturation can in fact be efficiently maintained, using so-called *shrinking sets*, as shown by, for example, Turner [16].

4.2 Obtaining the Maximum Clique

Algorithm 2 shows a simplified branch-and-bound approach to the Maximum-Clique problem. The algorithm relies partially on the calculation of an approximate coloring of the graph, which is used as an upper bound.

Algorithm 2 can be improved in a number of ways without jeopardizing optimality of the computed clique [7]. Let G be the graph at some point of the recursion, Q the clique under construction, $\widehat{Q}$ the current best solution, and $\{I_1, \ldots, I_k\}$ a k-coloring obtained on G. Then the following improvements apply:

- When we reach a state where $|Q| + |V(G)| \leq |\widehat{Q}|$, we can immediately prune the search space, since it is impossible to find a larger clique.
- Every vertex v such that $\delta(v) < |\widehat{Q}| - |Q|$ must be removed from the graph, because it can not be a member of a larger clique.
- Every vertex v such that $\delta(v) > |V(G)| - 2$, must be put into Q, since excluding it can not produce a larger clique.
- Every vertex v that can be colored with q colors, where $q > |Q| - |\widehat{Q}| + k$, yield unsuccessful branches, and can be left without further consideration.

The approximate coloring part of Maximum-Clique is a very simple greedy algorithm, using no particular heuristic for the ordering of vertices. The graphs have been represented simply as two-dimensional bit matrices. For the sake of efficiency, the graphs ought to have been redundantly represented as adjacency lists as well as matrices, a representation that has been chosen in register allocators ever since Chaitin's original algorithm. Our data structures are of course easy to extend to this double representation.

Algorithm 2 Find the maximum clique of a graph G using a simplified branch-and-bound technique. Return the set of vertices contained in the maximum clique found.

```
MAXIMUM-CLIQUE(G)
 1  return MAXIMUM-CLIQUE-RECURSIVE(G, ∅, ∅, ∞)
```

Input: G is the remaining part of the graph, Q is the clique under construction, $\widehat{Q}$ is the largest clique found so far, and u is an upper bound on ω (size of the maximum clique)

```
MAXIMUM-CLIQUE-RECURSIVE(G, Q, Q̂, u)
 1  if G is empty then
 2       return Q
 3  {I_1, ..., I_k} ← APPROXIMATE-COLOR(G)
 4  u ← MIN(u, |Q| + k)                        ▷ compute a new upper bound
 5  if u ≤ |Q̂| then
 6       return Q̂
 7  v ← a maximum degree vertex of G
 8  G′ ← subgraph induced by N[v]
 9  Q̂ ← MAXIMUM-CLIQUE-RECURSIVE(G′, Q ∪ {v}, Q̂, u)
10  if u = Q̂ then
11       return Q̂
12  G″ ← graph induced by V[G] − {v}
13  return MAXIMUM-CLIQUE-RECURSIVE(G″, Q, Q̂, u)
```

5 Experimental Results and Issues for Future Research

In the experiments with the graphs, using algorithms shown in Section 4, several interesting observations have been made. First and very importantly, *every single graph investigated turned out to be 1-perfect*, that is, for every single instance of the almost 28,000 graphs investigated, the chromatic number was determined to be exactly equal to the clique number.

The chromatic numbers of the Appel-George graphs range between 21 and 89. Most of them (27,590 graphs) have $\chi = 21$; 238 graphs have $\chi = 29$. Other test programs written, and compiled with the Cederberg compiler, get chromatic numbers on the interference graphs with a size of up to 15. All of them are 1-perfect. Despite numerous persistent tries, we have not managed to create one single program that results in a non-1-perfect interference graph using our compilers.

This experimental result raises two important questions to be further explored:

1. Are interference graphs always 1-perfect? Our experiments give strong empirical evidence for this. If it is the case, we need to determine why. That is, what in the earlier structural optimizations makes the graphs 1-perfect? Further graph sets from different (kinds of) compilers need to be examined in the future.

2. If all, or almost all, interference graphs are 1-perfect, how can we use this fact, in order to improve on the efficiency of the existing register allocation algorithms? Or should we rather incorporate qualities, such as simplification and/or copy propagation by live range coalescing, from the approximate algorithms into the exact algorithm?

The second question partially gets an answer through the second of our experimental results. We noted when running the exact algorithms on the graphs, that they seemed to be surprisingly fast. Hence, the George-Appel allocator was also implemented, using the pseudo-code given in [10], and the same data structures for graph representation as in the exact algorithms. Repeatedly running this algorithm for all the graphs, using an N-value equal to the chromatic number determined by the exact algorithm, and comparing execution times to those of the exact algorithm, gave the following result:

The exact algorithm for computing an optimal coloring is faster than George-Appel's approximate iteration algorithm.

Of course, the George-Appel allocator suffers a penalty through our choice of a data structure — the authors recommend a combination of bit matrices and adjacency lists.

But a change of the data structure would improve on the execution time of the exact algorithm as well. The operation for determining the neighborhood of a given vertex is expensive when using bit matrices, and it is very frequently used in both algorithms.

The execution times for the two algorithms have been plotted as functions of the sizes of the graphs in Fig. 1. One large and, for some reason, very tough, however, still 1-perfect graph instance containing 6,827 vertices, 45,914 edges, and 4,804 move related instructions has been removed from the data. (The time needed by the George-Appel algorithm to create the coloring was 3,290 seconds. The exact algorithm needed 120 seconds.)

In order to show the difference trend in the execution times of the two algorithms, a second degree polynomial has been fitted to the samples using the least-squares method. We do not, however, assert that the execution times are quadratic in the sizes of the graphs; the exact algorithm is obviously exponential in the worst case.

In Fig. 2 the same execution times are plotted for the 23,000 smallest graphs only, excluding the few very large and extremely tough instances.

There is one more thing which is important to note in the comparison of the two algorithm approaches. In 46 of the 28,000 graphs, the George-Appel algorithm fails to find optimum, and spills one or two variables to memory. This number of failures is actually impressively low, as the algorithm uses an approximate, heuristic method for the NP-complete problem of coloring. Perhaps the reason for the good performance of the approximate algorithm is the 1-perfectness of the graphs? Nevertheless, in comparison to the exact algorithm, these spills are of course unfortunate, especially since it does not seem to take longer time to find optimal colorings of the graphs using the exact algorithm.

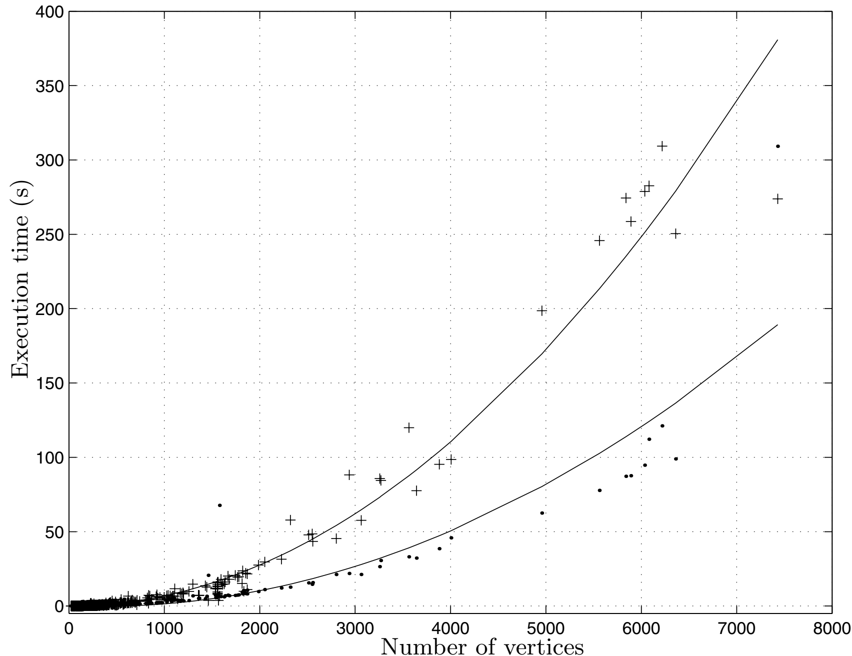

Fig. 1. Execution times for the two algorithms, coloring all graph instances. The dots correspond to the exact algorithm; the pluses correspond to the George-Appel allocator. The continuous functions are the best approximate second degree polynomials in the least-squares sense. We do *not*, however, assert that the execution times are quadratic in the sizes of the graphs. The intention is simply to compare the average execution times of the algorithms for the graphs in question.

6 Conclusions

Graph coloring is an elegant approach to the register allocation problem. The algorithms used by compilers today make use of approximate heuristics to accomplish the colorings.

In this paper, we do not propose a new algorithm for register allocation. The experiments, however, suggest that such an algorithm may well be designed, which guarantees optimal colorings for the purpose of a good allocation. Despite the fact that graph coloring is an NP-complete problem, the input graphs in the case of register allocation certainly seem to be efficiently colored, even when using an exact algorithm.

In the implementation of the sequential coloring algorithm, none of the typical improvements designed for register allocation, such as copy propagation by coalescing, graph simplification by vertex removal/merging, or interference reduction by live range splitting, have been accounted for. Our original purpose

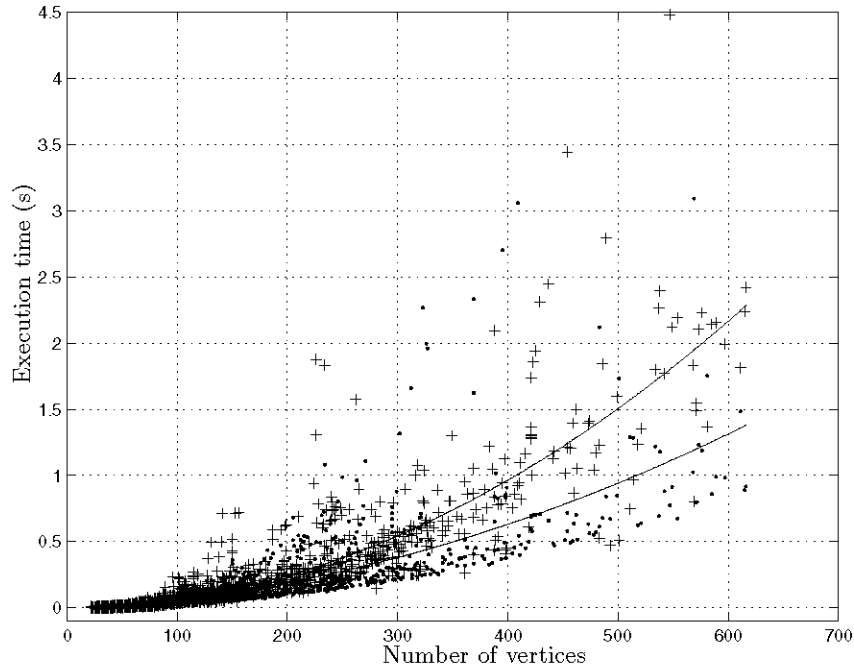

Fig. 2. Execution times for the two algorithms, coloring the 23,000 smallest graphs. The dots and the lower function estimate correspond to the exact algorithm; the pluses and the upper estimate correspond to the George-Appel allocator. The functions are second degree polynomials estimated from all the measured points using the least-squares method.

was simply to determine whether the graphs were 1-perfect, since this could have the effect of making optimal colorings efficiently computable.

In order to become applicable for register allocation, the algorithms need to implement such functionality. After all, most of the graphs investigated have much too large chromatic numbers, when not simplified, to fit into the register file of most processors. Even if the processor has a large register file, it is still desirable that programs do not use more registers than necessary, since loads and stores, made for instance at procedure calls, suffer most considerably when having to switch large numbers of registers into and out of memory.

In order to complete the goal of improving the register allocation algorithms, some questions remain to be answered:

- Do coalescing, merging, or splitting, the way we use these improvements in register allocators, jeopardize the 1-perfectness of the graphs?
- Is the 1-perfectness of interference graphs provable?
- Can we perhaps further strengthen the constraints in order to restrict the graph classes towards perfectness?

– How expensive does the exact coloring algorithm become if the graphs are not 1-perfect?
– Is it at all possible to implement an efficient register allocator that contains the different graph simplifications, and still guarantees the optimality of the produced colorings?

We believe that the answer to the last of these questions may well be positive, and our work will be continued with the goal of achieving such an implementation.

Acknowledgments. The author would like to thank

– *Dr. Jonas Skeppstedt* for his enthusiastic support in all parts of this work,
– *Per Cederberg* for lending us his experimental compiler implementation.

References

[1] Andrew W. Appel and Lal George. Sample graph coloring problems. Available online at `http://www.cs.princeton.edu/~appel/graphdata/`, 1996. 27,921 actual register-interference graphs generated by Standard ML of New Jersey version 1.09, compiling itself.
[2] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, April 1979.
[3] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
[4] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 1982. The Association for Computing Machinery.
[5] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, Montreal, June 1984. The Association for Computing Machinery.
[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, second edition, 2001.
[7] Olivier Coudert. Exact coloring of real-life graphs is easy. In *Proceedings of the 34th annual conference on Design automation conference*, pages 121–126, Anaheim, CA USA, June 1997. The Association for Computing Machinery.
[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.
[9] Lal George and Andrew W. Appel. Iterated register coalescing. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 208–218, St. Petersburg Beach, FL USA, January 1996. The Association for Computing Machinery.
[10] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[11] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer–Verlag, Berlin Heidelberg, Germany, 1988.
[12] Alfred Bray Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2:193–201, 1879.
[13] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.
[14] Luděk Kučera. *Combinatorial Algorithms*. Adam Hilger, Redcliff Way, Bristol BS1 6NX, England, 1990.
[15] Luděk Kučera. The greedy coloring is a bad probabilistic algorithm. *Journal of Algorithms*, 12(4):674–684, December 1991.
[16] Jonathan S. Turner. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9(1):63–82, March 1988.
[17] Steven R. Vegdahl. Using node merging to enhance graph coloring. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 150–154, Atlanta, GA USA, May 1999. The Association for Computing Machinery.
[18] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

A A Simple Example

Fig. 3 presents parts of a sample compiling session using the Cederberg compiler. First a high-level source code is input to the front-end. The intermediate representation produced by the front-end is presented to the right. For the purpose of simplicity, the user I/O functions, `get` and `put`, are assumed to be implemented as processor instructions.

Temporary variables in the program are named `t1`, `t2`, `t3`,...; basic blocks are labeled `x1`, `x2`, `x3`,....

The IR from the front-end is transformed into SSA form and is subject to two optimizations, *constant propagation with conditional branches* [18] and *partial redundancy elimination* [13], the result of which is shown down to the left on normal form, i.e., transformed back from SSA.

Analyzing the live ranges of the variables, inserting an edge between ranges that interfere, gives the interference graph presented down to the right. The graph has the maximum clique

$$Q = \{\mathtt{t11}, \mathtt{t16}, \mathtt{t17}, \mathtt{t21}, \mathtt{t22}\}, \quad \omega = 5,$$

and we conclude, directly from the figure, that the chromatic number χ is no larger than ω. Hence the graph is 1-perfect.

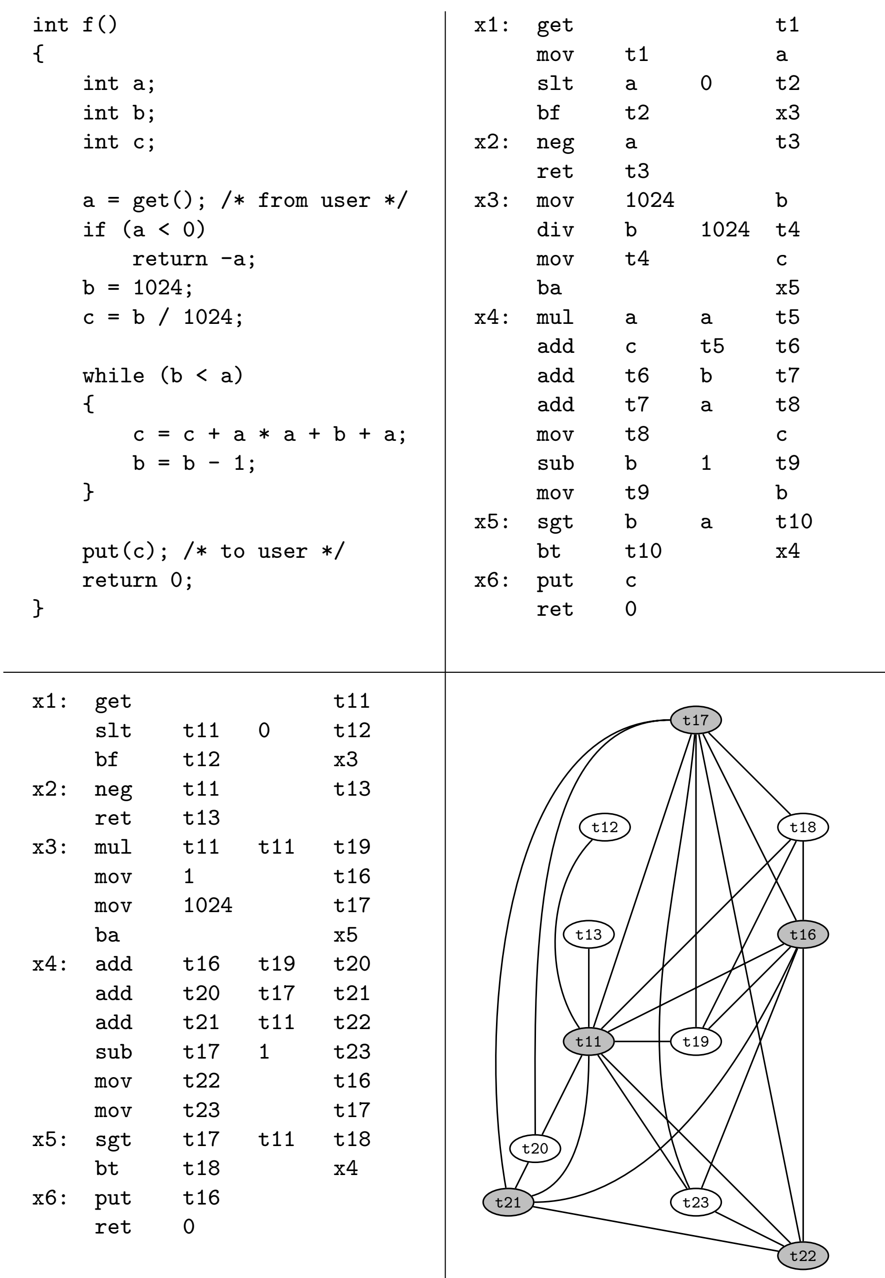

Fig. 3. *Top-left*: A high-level source code which is input to the compiler front-end. *Top-right*: IR output from the front-end. *Bottom-left*: The final improved IR from the optimizer. *Bottom-right:* IG with the maximum clique Q of size $\omega = 5$ shown with shaded vertices. Since, apparently, $\chi = \omega$ the graph is 1-perfect.